09/960,030

L Number	Hits	Search Text	DB	Time stamp
2	0	(computer and ("mathematical functions" same hardware)) and ((trigometric or sine or cosine) and logarithm and "dot product" and derivative)	USPAT	2003/06/21 18:09
1	67	computer and ("mathematical functions" same hardware)	USPAT	2003/06/21 18:45
3	1	(computer and ("mathematical functions" same hardware)) and "taylor series"	USPAT	2003/06/21 18:47
4	57	(computer and pipeline) and "taylor series"	USPAT	2003/06/21 19:08
5	0	(computer and grahics and pipeline) and branch$3	USPAT	2003/06/21 19:08
6	1247	(computer and graphics and pipeline) and branch$3	USPAT	2003/06/21 19:40
7	0	((computer and graphics and pipeline) and branch$3) and "greater than"	USPAT	2003/06/21 19:10
8	0	((computèr and graphics and pipeline) and branch$3) and (branch same "less than")	USPAT	2003/06/21 19:10
9	0	((computer and graphics and pipeline) and branch$3) and (condition same "less than")	USPAT	2003/06/21 19:11
10	0	((computer and graphics and pipeline) and branch$3) and (if same "less than")	USPAT	2003/06/21 19:11
11	473	((computer and graphics and pipeline) and branch$3) and branching	USPAT	2003/06/21 19:17
13	0	((((computer and graphics and pipeline) and branch$3) and branching) and "false operation") and (less-than or "less than")	USPAT	2003/06/21 19:12
14	12	(((computer and graphics and pipeline) and branch$3) and branching) and (set adj branch$3)	USPAT	2003/06/21 19:13
12	48	(((computer and graphics and pipeline) and branch$3) and branching) and "false operation"	USPAT	2003/06/21 19:16
15	29	(((computer and graphics and pipeline) and branch$3) and branching) and (operation near (call or return or floor or fraction or cosine))	USPAT	2003/06/21 19:26
16	23	(((computer and graphics and pipeline) and branch$3) and branching) and (operation adj (call or return or floor or fraction or cosine))	USPAT	2003/06/21 19:27
17	0	((computer and graphics and pipeline) and branch$3) and (operation adj (call and return and floor and fraction and cosine))	USPAT	2003/06/21 19:28
18	0	(computer and graphics and pipeline and branch$3) and (operation adj (call and return and floor and fraction and cosine))	USPAT	2003/06/21 19:28
19	0	(computer and pipeline and branch$3) and (operation adj (call and return and floor and fraction and cosine))	USPAT	2003/06/21 19:28
20	0	(computer and pipeline and branch$3) and (operation adj (call and return and floor and fraction))	USPAT	2003/06/21 19:29
21	0	(computer and pipeline and branch$3) and (operation adj (call and return and floor))	USPAT	2003/06/21 19:29
22	3	(computer and pipeline and branch$3) and (operation adj (call and return))	USPAT	2003/06/21 19:29
23	0	((computer and graphics and pipeline) and branch$3) and (operation same (branch and call and return and false and equal and true and no and move and light and distance))	USPAT	2003/06/21 19:43

24	50	((computer and graphics and pipeline) and branch$3) and (operation same (branch and call and return and false and equal and true and no and move))	USPAT	2003/06/21 19:43
25	1	(((computer and graphics and pipeline) and branch$3) and (operation same (branch and call and return and false and equal and true and no and move))) and (branch near operation)	USPAT	2003/06/21 19:43

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING

August 1992 (Vol. 18, No. 8)

pp. 657-673 • Working with Persistent Objects: To Swizzle or Not to Swizzle

J.E.B. Moss

Pointer swizzling is the conversion of database objects between an external form (object identifiers) and an internal form (direct memory pointers). Swizzling is used in some object-oriented databases, persistent object stores, and persistent and database programming language implementations to speed manipulation of memory resident data. The author describes a simplifying model of application behavior, revealing those aspects where swizzling is most relevant in both benefits and costs. The model has a number of parameters, which the authors have measured for a particular instance of the Mneme persistent object store, varying the swizzling technique used. The results confirm most of the intuitive, qualitative tradeoffs, with the quantitative data showing that some performance differences between schemes are smaller than might be expected. However, there are some interesting effects that run counter to naive intuition, most of which are explained using deeper analysis of the algorithms and data structures.

Index Terms-pointer swizzling; persistent objects; database objects; object identifiers; direct memory pointers; object-oriented databases; persistent object stores; database programming language implementations; manipulation; memory resident data; Mneme persistent object store; quantitative data; data structures; data structures; object-oriented databases; software engineering

ENEE 647: Design of Distributed Computer Systems

Week 03.5 -- Virtual Memory Mechanisms

Papers

- Jacob 98: "Virtual memory: Issues of implementation"
- Jacob 98: "Virtual memory in contemporary microprocessors"
- Jacob 98: "A look at several memory management units, TLB-refill mechanisms, ...
- Khalidi 93: "Virtual memory support for multiple page sizes"

Sidetrack: Swizzling
---

since several people were interested in SWIZZLING, i figured
that i would take a little time to explain what it means.

swizzling was originally invented to implement PERSISTENT DATA STORES

(permanent virtual memory)

how can you make memory permanent?
as soon as you turn off the computer, all memory values are lost.

BACK THE MEMORY WITH THE DISK SYSTEM

problem -- if the region has pointers in it, how are they interpreted?
pointers on disk are meaningless -- the pointers are only valid if
EVERYBODY who uses the shared data AGREES on where it should go in their addr

(this is often considered an unrealistic expectation)
(however, if you implement a system this way, it makes your life a lot easier

anyway, INSTEAD OF POINTERS, when you store the object out to disk, use INVALID P

(what?)

instead of storing the following structure to disk exactly as it looks in memory:

```
struct node {
    struct node *left;
    struct node *right;
    char key[16];
    char data[128];
}
```

you first REPLACE each of the POINTERS with INVALID POINTERS
(here's an example to replace the pointers of a DAG within a 1MB shred region)

```
node *dag_root;         /* by convention, a local var pointing to start of sh
relativify(dag_root);
```

- first, determine the RELATIVE OFFSET of the object being pointed to

```
void
relativify(np)
struct node *np;
{
    struct node *tmpleft = np->left;
    struct node *tmpright = np->right;
    np->left -= dag_root;
    np->right -= dag_root;
```

- then, set the topmost bits so that we are guaranteed to cause a SEGMENTATIO

```
            np->left != 0xbad00000;
            np->right != 0xbad00000;

        - then update the chilldren pointers:

            relativify(tmpleft);
            relativify(tmpright);

            return;
        }
```

now, every pointer has been changed to a relative offset within the shared region
(assuming the relative region is no larger than 1MB)

at this point, we can simply copy the entire shared region onto the disk.

at a later point, another process (or a new invocation of this process) reads in
it is copied into a 1MB region, verbatim.

the local node *dag_root points to the first item in the shared region (by conven
and we begin operating on the data.

the first use of a pointer causes a SEGMENTATION VIOLATION because the top bits a
(most hardware systems disallow access to the top of the address space)

the hardware raises an interrupt, saves state, and vectors to SEGMENTATION VIOLAT

the operating system wakes up in the SEGMENTATION VIOLATION interupt handler.
it sends a SIGSEGV signal to the offending process.
this does not immediately kill the process -- it calls a user-specified routine t

the OS calls the signal handler coresponding to a SIGSEGV signal
NORMALLY, there is a die() handler attached to SIGSEGV -- by default, we die.
however, we can replace die() with swizzle()

```
    swizzle()
    {
        are the top bits of offending pointer "0xbad" ???
        if so, replace pointer with
            pointer & 0x000fffff + dag_root;
        and retry the load/store operation

        if not, die()
    }
```

now, pointer by pointer, we can replace all the bad pointers with the correct poi
alow but effective.

also -- does not need any operating system support.

Virtual Memory

why? what does it do for us?
- protection
- hardware independence
- a nice programming paradigm (the virtual machine)
- can execute program as soon as first page is resident

how? what does the implementation depend on?
it is actually a whole lot like swizzling, but this time the OS knows about i
- you use addresses to reference things (both code and data)
- the hardware tries to use them as-is
- if it can TRANSLATE the address, it does and uses it
- if it cannot, it raises an interrupt and calls the operating system

the operating system, if called, looks in its database: the PAGE TABLE
it finds the VIRTUAL->PHYSICAL mapping for the address (at a PAGE granularity
and gives that translation info to the hardware, which retries the operation

note that you do not NEED hardware support for VM to have VM;
it is just more efficient with hardware support

you should have learned about VM in your previous OS classes.
if not, read the first half of the Computer article to get caught up
(Jacob 98: "Virtual memory: Issues of implementation")

what you probably were NOT taught in your OS class is that there are MANY ways of implementing a page table, including the following:

top-down hierarchical
bottom-up hierarchical
inverted/hashed table
software TLB (a PTE cache)

the mechanics of these are described in the Computer article.

Jacob 98: Computer
---

rough look at hierarchical vs. inverted pages tables (H vs. I):
- H allocates space in the table 1 PAGE at a time
- I allocates space in the table 1 PTE at a time
- H table has BOUNDED lookup time
- H table has relatively UNBOUNDED lookup time
  (usually bounded by size of table, which is big)
- H table is COMPLETE -- holds all mapping information for all pages
- I table is INCOMPLETE -- only holds mapping information for pages in memory
  (need alt. structure to hold info for swapped pages, etc.)
- H table SUPPORTS ALIASING (multiple virtual addresses map to the same physi
- I table does not, unless the 1:1 mapping of PTEs to physical pages is elimi
  then each PTE must store VPN and PFN ... increases size of PTE and thus tab
  (starts to look like a regular database index structure)

rather than discuss the mechanics of the VM structures, i'll give the highlights:

top-down hierarchical is used in the x86 ... it doesn't scale too well to lar
because each level HAS to be traversed in a lookup
(however, see the guarded page table of Liedtke ...)

bottom-up hierarchical is used in the MIPS architecture ... MOST of the time,
you only look at one level of the table
(then resort to top-down if the initial access fails)

this has better performance than the top-down approach.
it also scales better to a 64-bit address space than the top-down approac
(e.g. it is used in Alpha -- four-tier page table)

original complaint: hierarchical tables are bad
- they are allocated one PAGE at a time -- do not support sparse address sp
- they require a memory access for every level of the table -- do not scale
the first argument is pretty solid.
the MIPS style table addresses the second (as does the guarded page table)

inverted table used by HP -- it is a hash table.
space in hash table allocated a PTE at a time -- THIS addresses the argum
the hierarchical tabls that it wastes memory.
you have as many PTEs as there are PAGE FRAMES in the system

however -- this means that the table is just a CACHE for ALL the mapping
where do we keep info for the pages kept on disk?

```
            one argument: keep that stuff in a smaller table, or keep it on disk.
            if the pages are non-resident, who cares how long it takes to retriev

        if we choose a bad structure for this, we can either eat up as much SPACE
            hierarchical table (to maintain the mappings for currently-non-reside
        OR: we might have something that is very slow to access.

        NOTE -- this is an open research topic

        the table is a big hash bucket -- you hash the VPN and index the table, t
            down a list of poitners to find the mapping information.
        the linked lists are kept short by there being a big hash table

        problems:
          - it can have slightly poorer performance relative to a hierarchical ta
          - the original design does not allow shared mappings to physical pages

    an explicit PTE cache: like in the Bala94 paper.
    as long as you have resigned yourself to keeping just a CACHE for PTEs around
        a mechanism that might be slow?
        have a direct-mapped cache, instead of a hash bucket.
        implemented in software:

            PTE ptecache[65536];    /* keeps 65K PTEs around */

        if the item is not found in the cache, look it up in the real page table.

there are a number of implementation issues that can cause great headache

    these are the big topics:
      - shared memory & virtual caches
      - shared memory & page table design
      - protection scheme and hardware support
      - TLB management
      - support for 64-bit addressing (and larger)

    shared memory is very dependent on the protection scheme:
        if the address spaces are protected via ASIDs, you have to share by
            circumventing the protection mechanism: aliasing
        if the address space uses SEGMENTATION, sharing is easy, however, it is
            still nice to use aliasing to do other things, like
            map the same shared region with different protections or
            attach different behaviors to each mapping
              - memory-consistency protocols
              - faulting/non-faulting access

        the problem with ALIASING is that it really messes with a virtual cache
          - segmentation can help solve the problem (see the Jacob 97 tech report
        another problem: if the page is remapped (swapped in or out), EVERY PTE t
            the page must be updated.
            this management can be overwhelming
        also: each mapping requires its own TLB entry -- performance degradation

        one solution: share PTEs whenever PAGES are shared.
        easy to do in a top-down hierarchical table (and any GLOBAL table)
          - this solves the management problem and the multiple TLB-entry problem
          - it also allows you to place shared regions at different virtual locat
          - it does NOT allow you to have different behaviors or protections atta
            to a region
            (this can be solved by segmented page table)

    TLBs can be managed in hardware or software (the ASPLOS paper quantifies this
        software: more flexible, lower performance
        hardware: less flexible, greater performance
```

PA-RISC: did a hybrid approach, where hardware checks the hash bucket's i
value -- th rest of the chain is walked in software if it is a miss

TLB flushing is required on context switch
- if you have no ASID in the TLB entry AND there is no other mechanism
(i.e. x86 is typically flushed -- but not necessary if segmentation is
- if you have shared pages that you mark GLOBAL in order to share across
- whenever an ASID is reassigned (there are finite ASIDs, many processes;
will often remap a process to an ASID many many times ...)

Jacob 98: Micro
---

the interaction between the OS & HW is best seen in the VM system, because HERE i
all of the assumptions (on both sides of the interface) are exposed

what is most interesting is the variation in support for what is essentially a ve
simple operation: translation and protection of virtual address spaces

numerous other papers give testimony to the problems of failing to match an OS to
the problem with VM hardware is that it is completely incompatible and requires
a LOT of OS tuning to exploit correctly -- it actually goes beyond TUNING, be
that suggests all you need to do is "tweak" the OS here & there and all will

i will not give a description of how each mechanism works.
instead, i will point out some interesting features and their implications

hardware-managed TLBs don't bother the pipeline -- they do not cause an interrupt
they do not flush the pipeline, they do not stomp on the I-cache.
they therefore have EXCELLENT performance -- this may make up for their infle
(see the ASPLOS paper)

perhaps the best design: hardware-managed TLB where the functional unit is dr
by a programmable FSM?
(see the ASPLOS paper)

the pentium pro (P6 and beyond) treats the TLB as any ordinary functional unit;
i.e. the system does not block on a TLB miss ... only the instruction that
depends on the TLB access blocks (and, of course, any instructions that depen
it would be a considerable amount of work to do this in a software-managed TLB
(note: we're working on exactly this problem)

MIPS has small ASIDs ... whenever you remap an ASID you flush the TLB
so -- every 256 processes, the TLB must be flushed (typically, even more often)

MIPS has a nice facility that builds the virtual address of the PTE for you.
therefore, your handler need only LOAD the pte -- you don't have to build its
the SPARC has a similar facility (that is not described in the article because at
time it was not public knowledge) -- they have a PTE cache (described up abov
hardware will build you an address that references a location in that cache.

note that many of the 64-bit systems out there today (mips, alpha and ultrasparc)
recognize all 64 bits.

segmentation: an interesting alternative to ASIDs (see the tech report)
they do a much better job of organizing the big global space.
ASIDs chunk it into a bunch of address spaces -- segmentation chunks it into
bunch of address space COMPONENTS -- pieces that MAKE UP an address space
this (arguably) is a better way to implement a system -- IF THERE IS SHARING
(otherwise, just use ASIDs)

segments act as organization and protection -- provided their mapping is protecte
the PowerPC makes the segment-mapping a protected mechanism
the PA-RISC does not
the PowerPC therefore needs no additional protection mechanism
the PA-RISC does need additional protection

interesting thing about segments -- if done right, never need to flush the TLB
(well, not quite NEVER, but RARELY)

PA-RISC separates NAMING from PROTECTION
the segmentation mechanism is used to NAME things -- it does NOT provide protecti
(as it does in PowerPC and x86)
this allows processes full access to a 96-bit virtual address space (except for t
regions that it cannot touch)
other mechanisms do not allow you to EXTEND your address space -- PA-RISC does

what are the advantages?
i'll not tell you -- you might want to do this for a semester project.
(the proof is left to the reader ...)

what do people do to avoid having to rewrite operating system code?
they define Hardware Abstraction Layers (Mach, NT, etc)
these HIDE the hardware from the operating system - only a small amount of code
actually touches the hardware
the interface:
create_map
destroy_map
protect_page
protection_region
wire_down_page
wire_down_region
etc.
then the OS is free to implement whatever it wants on top of this.
problem: these building blocks are almost entirely useless.
they form the most basic things that virtual memory does.
the operating system, to get good performance, makes a number of decisions at a h
that are much more subtle, and which are hardware-specific:

virtual vs. physical caches -> consistency management issues
size of ASID (big/small) -> frequency of TLB flushing
protection methods available -> organization of shared memory code
the shared memory interface
the methods for object allocation/destruction

the list goes on.

the HARD things to encapsulate are the things that cause all the problems.

this is an important point, so i will say it three more times.

the HARD things to encapsulate are the things that cause all the problems.
the HARD things to encapsulate are the things that cause all the problems.
the HARD things to encapsulate are the things that cause all the problems.

this is almost EXACTLY what the first paper we read was saying (Waldo, et al), ex
they were saying it about distributed systems

it is a catch-22: if something is easy to encapsulate, it is probably not necessa

Jacob 98: ASPLOS
---

virtual memory interfaces are HIGHLY proprietary ... among other things,
they guarantee that system software is COMPLETELY non-portable.

why? is there a performance benefit to choose one over another?
surprisingly, there has never been a study. this is the first of its kind.

the study quantifies the following (some of which is obvious, but had never been

- a hardware-walked page table performs very well:

- it doesn't step on the i-cache
- it doesn't cause an interrupt

as a result, it can make a mediocre page table design (like that of the x86)

- an inverted page table impacts the cache less than other types:
  - PTEs are VERY unlikely to be unused and therefore
  - the PTEs are clustered together in space

this, even if the individual PTEs are much larger than in the other page tabl

- the VM overhead, when all is said & done, is almost three times what was thou
  - typical TLB overhead: 5-10% of runtime
  - caching effects of page table add another 5-10%
  - interrupts can add another 5-10%

- need to design a better interrupt model
  (present one: wait until instruction is at head of reorder buffer, then flush

PRECISE STATE

what is precise state? the state that the machine WOULD HAVE BEEN IN,
had we been executing instructions sequentially
very restrictive definition

how do you do it with out-of-order execution?
very carefully.

essentially: you have each instruction keep track of the machine's state imme
before it (previous register contents), so that it can be UNDONE
then, if an instruction causes an exception,
each partially-completed instruction is UNDONE in REVERSE ORDER
therefore, when you are done, you have the PRECISE STATE as the machine w
at the START of the FIRST PARTIALLY-COMPLETED INSTRUCTION

(draw this)

typical implementation:
MARK instruction as EXCEPTION-CAUSING
once it is NEXT-TO-COMMIT, undo all partially-completed instructions
(every instruction after this one)
THEN, once pipe is flushed, jump to exception handler code

we wait until the instruction is NEXT-TO-COMMIT -- this guarantees that n
will be handled out-of-order (which would be against the definition o

problem: we end up flushing DOZENS of instructions, and then RE-EXECUTE them

anyway ... this study looks at a few combinations of hardware and software

A few of the main conclusions:

- if you have TLBs, they had better be big

- as systems get bigger (more concurrency), interrupts increase in cost to the
  where they completely dominate all other costs in memory management

- cost of interrupts scales with TLB size, not cache size
  (unless you are using a software-managed cache design)

- might want to do a study of page table org's to see their cache behavior
  and how they interact with user code (this study just takes a look at
  total execution time, which only tells you so much ...)

- the base overhead of TLB handler and page table org. yields relatively little
  real systems DO NOT look like these stripped-down implementations ... they ha
  of kruft code that weighs down the system

- superpages will solve much of the problem, but they are difficult to use

- might want to investigate a configurable state-machine approach to page table

- software-managed caches are viable design option

- in general, all the systems performed roughly the same -- why have so many di
  HW implementations? why not standardize on some interface & go from there?

Khalidi 93
---

here are some more assumptions that SW makes about HW:

- there is one page size
- physical memory is relatively small

TLB maps relatively small amount of memory:

128-entry TLB with 8KB pages maps 1MB

this is smaller than most L2 CACHES

working sets are growing larger faster than TLBs are growing -- TLB maps smaller
amount of space

"TLB REACH"

as a result, much hardware now supports MULTIPLE PAGE SIZES

works well for very specialized applications

how to get it to work for general applications?

hard, because OS typically designed for ONE PAGE SIZE

FIGURE 1: if we caould actually TAKE ADVANTAGE of 32KB page sizes, we would REDUC

what needs to be done?

- who chooses the page size?
  probably the OS -- export the abstraction to the compiler as a HINT

- the KEY used to look up mapping information changes:
  the VPN and PFN have to change to be the LEAST COMMON DENOMINATOR
  (page numbers for the BIGGEST POSSIBLE SIZE, and possibly map to multiple PTE

- have to manage both physical and virtual buffer lists.
  painful. have to deal with fragmentation, page replacement on multiple level

- when to choose between large/small page?
  dynamic PROMOTION is their technique

how do we do page replacement?
currently, an approximation to LRU:

currently, we keep a REFERENCED bit for every page in the system.
periodically, CLEAR every REFERENCED bit
later, if we come through and the bit is still CLEAR, it has not been referen
MARK THAT PAGE FOR POSSIBLE REPLACEMENT.

as we go to larger memory systems, this clearly doesn't work.
1GB of physical memory == ~100 thousand pages
10GB of physical memory == ~1 million pages

the OS would spend almost all of its time going through the REFERENCED bits

perhaps RANDOM?
perhaps FIFO?